

Trabajo 3

Antonio Álvarez Caballero

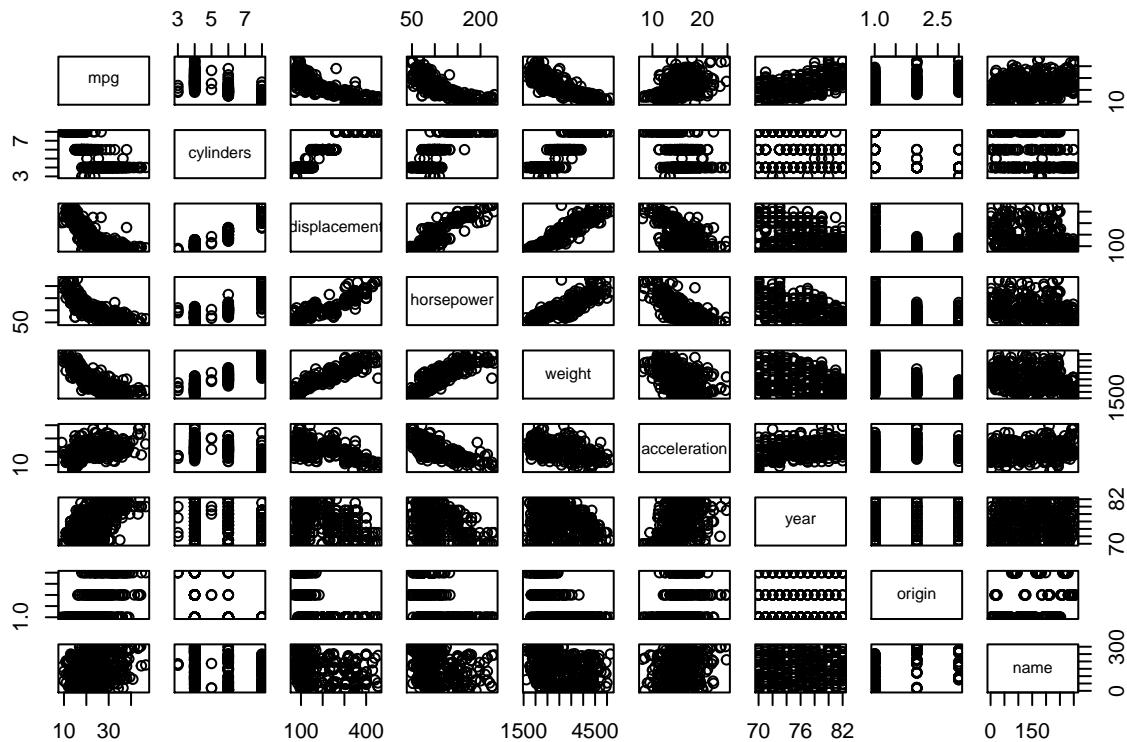
27 de mayo de 2016

Ejercicio 1

```
## The following object is masked from package:ggplot2:
```

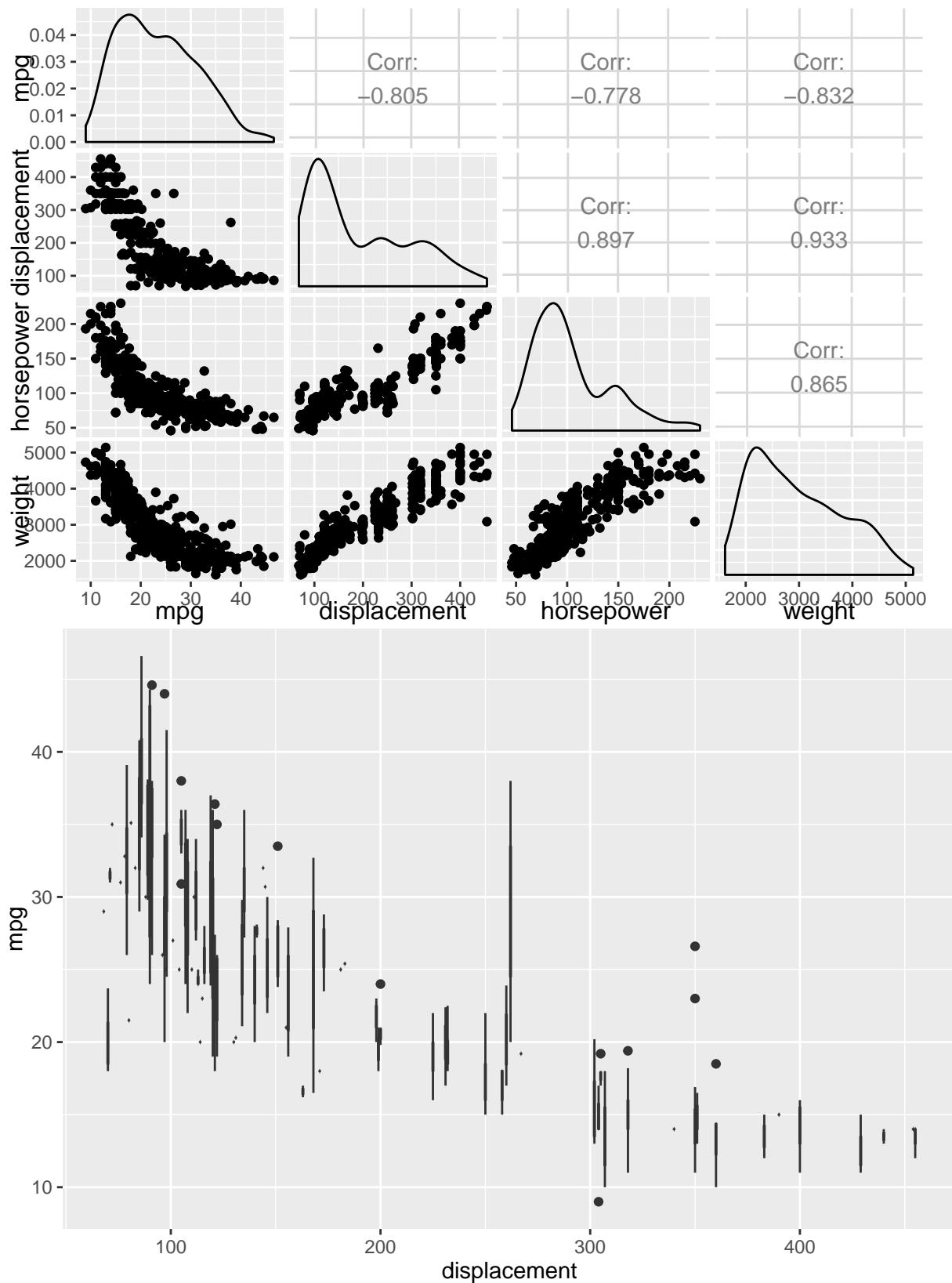
```
##
```

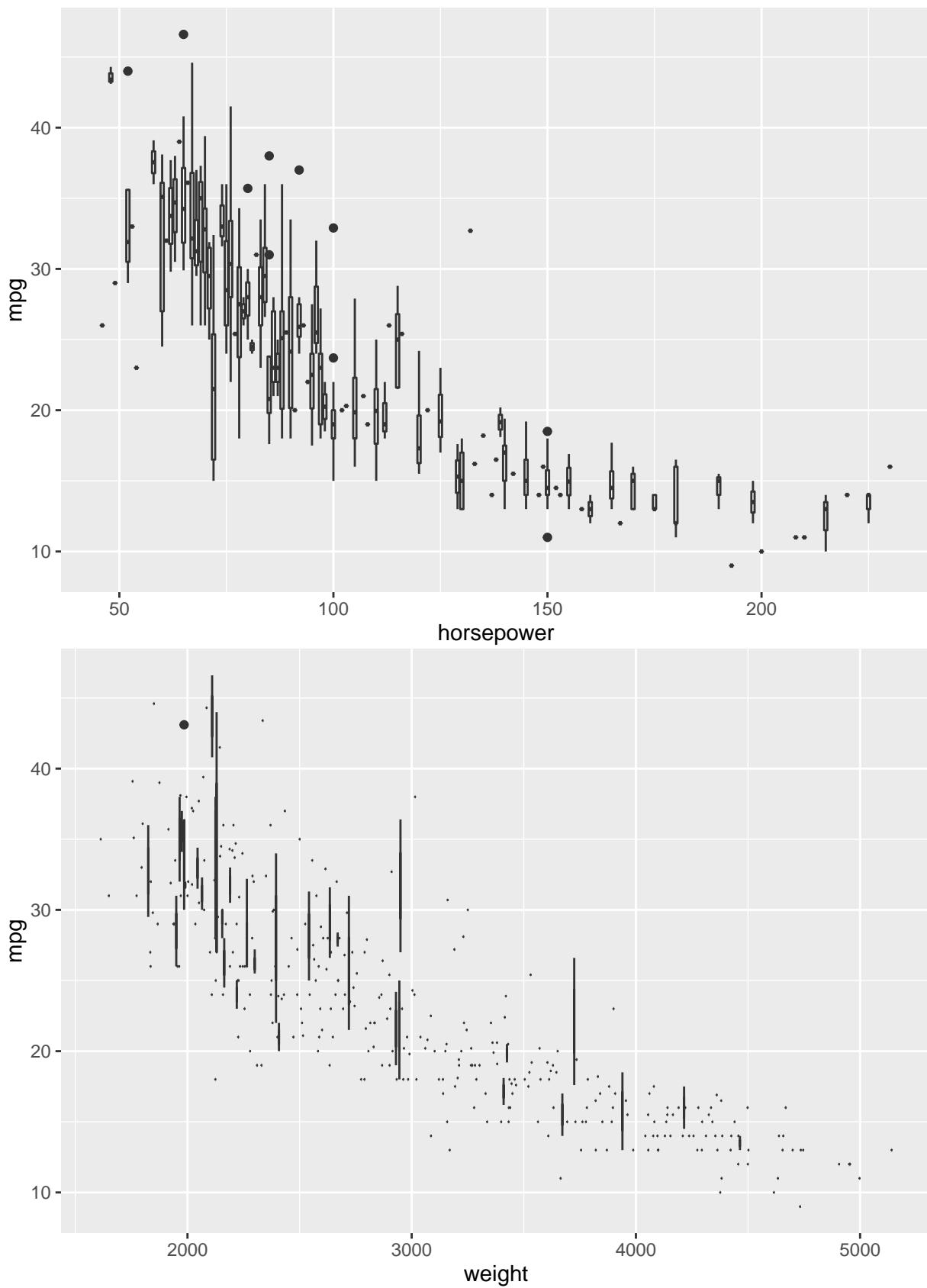
```
##     mpg
```



Apartado a)

Parece ser que las variables de las que más depende *mpg* son *displacement*, *horsepower* y *weight*. Veámoslas con más detalle.





Es claro que existe una dependencia entre estas variables. Además, con el primer plot podemos ver las

correlaciones entre estas 3 variables, que es alta.

Apartado b)

Seleccionamos las variables que hemos decidido para predecir.

```
Auto.selected <- Auto[,c("displacement","horsepower","weight")]
```

Apartado c)

Como nuestro conjunto de datos es grande (392 instancias), podemos realizar un muestreo aleatorio. Así tampoco falseamos las muestras, cosa que podría pasarnos si realizamos un muestreo estratificado.

```
index <- sample(nrow(Auto), size = 0.8*nrow(Auto) )  
  
Auto.train <- Auto.selected[index,]  
Auto.test <- Auto.selected[-index,]
```

Apartado d)

Vamos a crear una nueva variable, *mpg01*, la cual tendrá 1 si el valor de *mpg* está por encima de la mediana y -1 en otro caso.

```
mpg01 <- ifelse(Auto$mpg >= median(Auto$mpg), 1, 0)  
Auto.selected$mpg01 <- mpg01  
Auto.train$mpg01 <- mpg01[index]  
Auto.test$mpg01 <- mpg01[-index]
```

Apartado d1)

Vamos a ajustar un modelo de regresión logística para predecir *mpg01*.

```
model.LogReg <- glm(mpg01 ~ ., data = Auto.train, family = binomial)  
prediction.LogReg <- predict(model.LogReg, newdata = Auto.test, type = "response")  
prediction.LogReg.labels <- ifelse(prediction.LogReg > 0.5, 1, 0)  
error.test <- sum(sign(prediction.LogReg.labels) != sign(Auto.test$mpg01)) /  
length(prediction.LogReg.labels)
```

El error de test de este modelo es 8.8607595.

Apartado d2)

Ahora vamos a ajustar un modelo k-NN.

Apartado d3)

Veamos las curvas ROC de ambos modelos.

```

roc.prediction.regLog <- prediction(prediction.LogReg, Auto.test$mpg01)
roc.performance.regLog <- performance(roc.prediction.regLog, measure = "tpr", x.measure = "fpr")

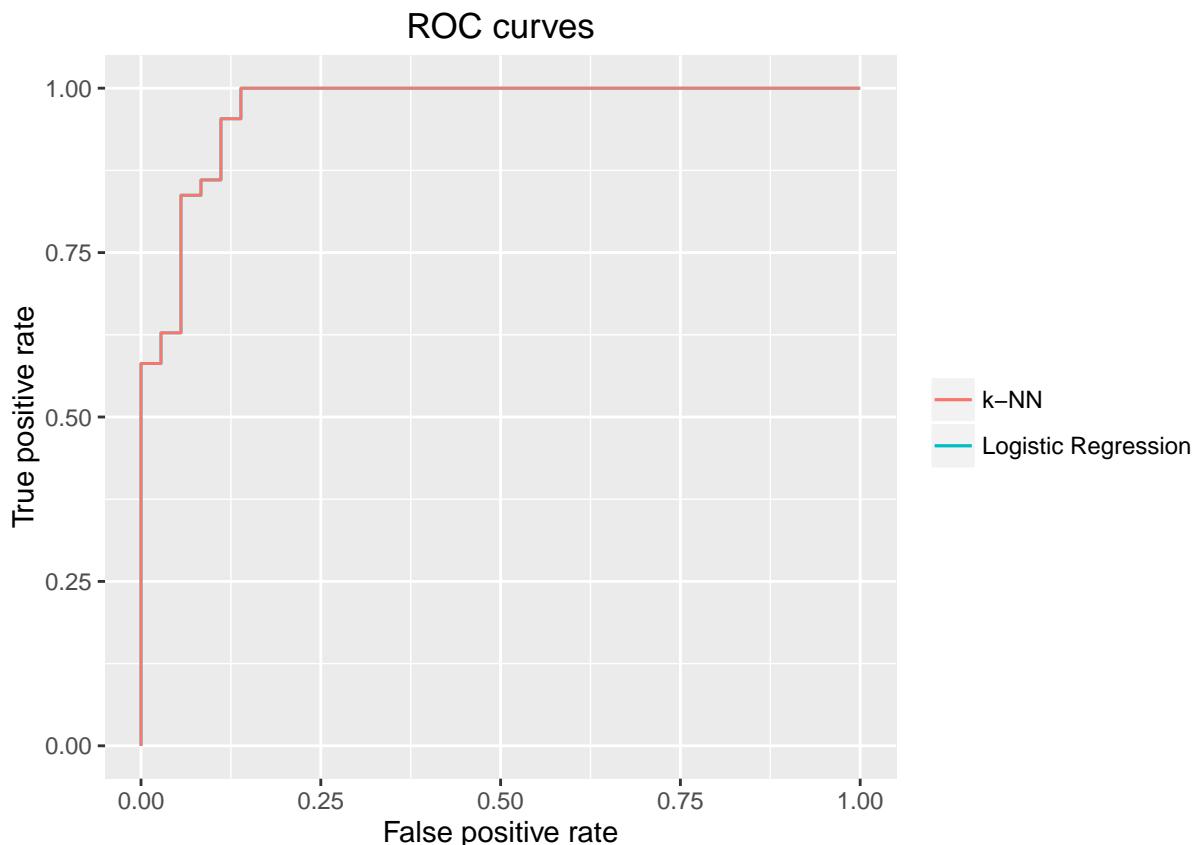
### Meter del knn

#####
roc.performance.knn <- roc.performance.regLog

roc.data <- data.frame(x = roc.performance.regLog@x.values[[1]],
                       y1 = roc.performance.regLog@y.values[[1]],
                       y2 = roc.performance.knn@y.values[[1]])

ggplot(roc.data, aes(x)) +
  geom_line(aes(y = y1, colour = "Logistic Regression")) +
  geom_line(aes(y = y2, colour = "k-NN")) +
  theme(legend.title = element_blank()) +
  labs(title= "ROC curves", x = "False positive rate", y = "True positive rate")

```



```

auc.regLog <- auc(roc.data$x,roc.data$y1, type = 'spline')
auc.knn <- auc(roc.data$x,roc.data$y2, type = 'spline')

```

El área bajo la curva de la ROC de regresión logística es 0.9651519 y la del k-NN es 0.9651519. Luego k-NN es el modelo que mejor *performance* tiene.

Apartado e) (Bonus-1)

Para estudiar el error con validación cruzada hacemos uso de `cv.glm`

```
model.full.LogReg <- glm(mpg01 ~ ., data = Auto.selected)
cv.LogReg <- cv.glm(data = Auto.selected, glmfit = model.full.LogReg, K = 5)
cv.LogReg$delta
```

```
## [1] 0.1043928 0.1040979
```

El error estimado es el primero de este vector. El segundo es un ajuste para compensar el sesgo introducido al no usar *Leave-One-Out*.

Para el caso del k-NN

Por tanto, vemos que es mejor *uno*.

Apartado f) (Bonus-2)

Por hacer

Ejercicio 2

Apartado a)

Ajustamos con validación cruzada sobre la variable *crim*, que es la que está en la posición 1.

```
attach(Boston)

set.seed(123456789)
index <- sample(nrow(Boston), 0.8*nrow(Boston))
Boston.full <- Boston
Boston.train <- Boston[index,]
Boston.test <- Boston[-index,]

model.Boston <- glmnet(as.matrix(Boston.train[,-1]), Boston.train[,1], alpha = 1)
```

Apartado b)

Ahora utilizamos un método LASSO y seleccionamos las variables que están por encima de un umbral.

```
cv.Boston <- cv.glmnet(as.matrix(Boston[,-1]), Boston[,1], nfolds = 5, alpha = 1)
lasso.coeff <- predict(cv.Boston, type="coefficients", s = cv.Boston$lambda.min)
threshold <- 0.1
selected <- which(abs(lasso.coeff) > threshold)[-1]
```

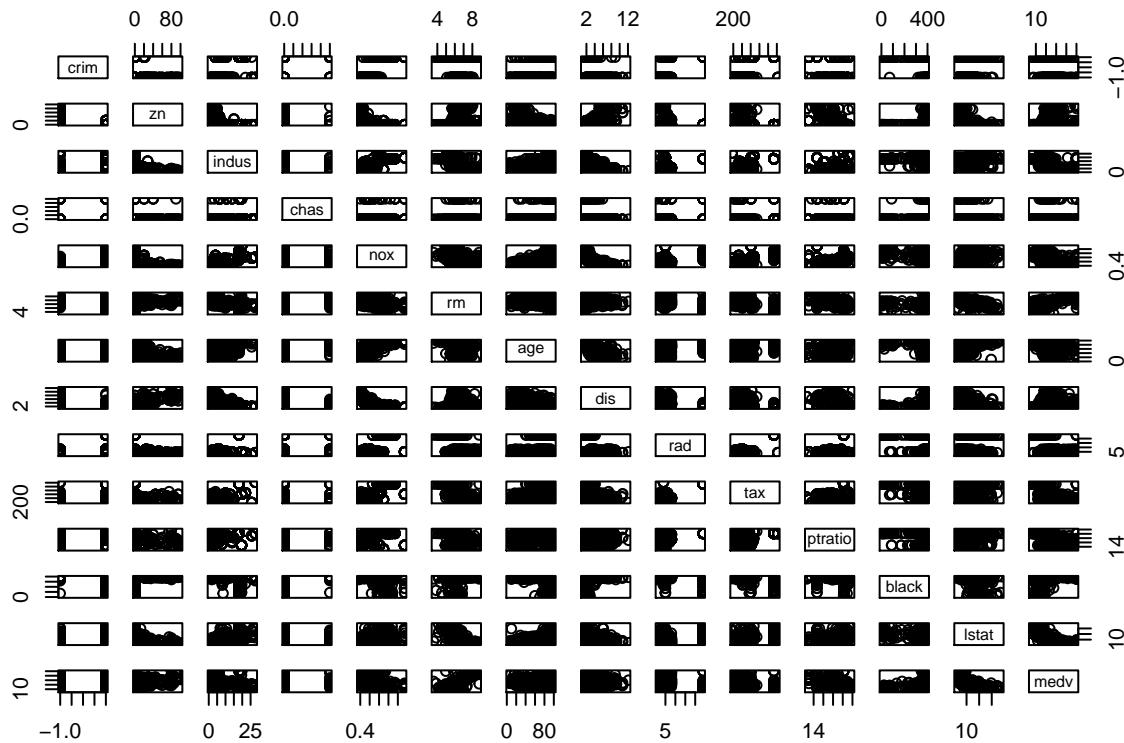
Con esto afirmamos que las características que superan nuestro umbral 0.1 son 4, 5, 6, 8, 9, 11, 13, 14.

Seguir con regularización.

Apartado c)

Al igual que en el anterior apartado, definimos una nueva variable usando la mediana como umbral.

```
Boston.full$crim <- ifelse(Boston$crim > median(Boston$crim), 1, -1)
Boston.train.crim1 <- Boston.full[index,]
Boston.test.crim1 <- Boston.full[-index,]
pairs(Boston.full)
```



Ahora ajustamos varias *SVM*, probaremos la lineal y con los núcleos disponibles, y veremos cómo se comporta cada uno. Posiblemente el núcleo lineal no sea suficiente, ya que los datos no parecen lo suficientemente separados. A priori parece que uno polinomial hará un mejor trabajo.

```
svm.linear <- svm(crim ~ ., data = Boston.train.crim1, kernel = "linear")
svm.linear.prediction <- predict(svm.linear, newdata = Boston.test.crim1[,-1])
confusionMatrix(sign(svm.linear.prediction), Boston.test.crim1$crim)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction -1   1
##           -1 51 10
##            1   3 38
##
##                 Accuracy : 0.8725
##                 95% CI : (0.7919, 0.9304)
##     No Information Rate : 0.5294
##     P-Value [Acc > NIR] : 1.616e-13
##
##                 Kappa : 0.7421
##  Mcnemar's Test P-Value : 0.09609
##
##                 Sensitivity : 0.9444
##                 Specificity : 0.7917
##     Pos Pred Value : 0.8361
```

```

##           Neg Pred Value : 0.9268
##           Prevalence : 0.5294
##           Detection Rate : 0.5000
##   Detection Prevalence : 0.5980
##           Balanced Accuracy : 0.8681
##
##           'Positive' Class : -1
##
svm.polynomial <- svm(crim ~ ., data = Boston.train.crim1, kernel = "polynomial")
svm.polynomial.prediction <- predict(svm.polynomial, newdata = Boston.test.crim1[,-1])
confusionMatrix(sign(svm.polynomial.prediction), Boston.test.crim1$crim)

## Confusion Matrix and Statistics
##
##           Reference
## Prediction -1  1
##           -1 52  8
##           1   2 40
##
##           Accuracy : 0.902
##           95% CI : (0.8271, 0.952)
##   No Information Rate : 0.5294
##   P-Value [Acc > NIR] : 5e-16
##
##           Kappa : 0.8019
##   Mcnemar's Test P-Value : 0.1138
##
##           Sensitivity : 0.9630
##           Specificity : 0.8333
##   Pos Pred Value : 0.8667
##   Neg Pred Value : 0.9524
##           Prevalence : 0.5294
##           Detection Rate : 0.5098
##   Detection Prevalence : 0.5882
##           Balanced Accuracy : 0.8981
##
##           'Positive' Class : -1
##
svm.radial <- svm(crim ~ ., data = Boston.train.crim1, kernel = "radial")
svm.radial.prediction <- predict(svm.radial, newdata = Boston.test.crim1[,-1])
confusionMatrix(sign(svm.radial.prediction), Boston.test.crim1$crim)

## Confusion Matrix and Statistics
##
##           Reference
## Prediction -1  1
##           -1 49  6
##           1   5 42
##
##           Accuracy : 0.8922
##           95% CI : (0.8152, 0.9449)
##   No Information Rate : 0.5294
##   P-Value [Acc > NIR] : 3.774e-15

```

```

##                                     Kappa : 0.7833
##   Mcnemar's Test P-Value : 1
##
##                                     Sensitivity : 0.9074
##                                     Specificity : 0.8750
##   Pos Pred Value : 0.8909
##   Neg Pred Value : 0.8936
##   Prevalence : 0.5294
##   Detection Rate : 0.4804
##   Detection Prevalence : 0.5392
##   Balanced Accuracy : 0.8912
##
##   'Positive' Class : -1
##
svm.sigmoid <- svm(crim ~ ., data = Boston.train.crim1, kernel = "sigmoid")
svm.sigmoid.prediction <- predict(svm.sigmoid, newdata = Boston.test.crim1[,-1])
confusionMatrix(sign(svm.sigmoid.prediction), Boston.test.crim1$crim)

## Confusion Matrix and Statistics
##
##   Reference
## Prediction -1  1
##   -1 22 18
##   1 32 30
##
##   Accuracy : 0.5098
##   95% CI : (0.4089, 0.6101)
##   No Information Rate : 0.5294
##   P-Value [Acc > NIR] : 0.69044
##
##   Kappa : 0.0319
##   Mcnemar's Test P-Value : 0.06599
##
##   Sensitivity : 0.4074
##   Specificity : 0.6250
##   Pos Pred Value : 0.5500
##   Neg Pred Value : 0.4839
##   Prevalence : 0.5294
##   Detection Rate : 0.2157
##   Detection Prevalence : 0.3922
##   Balanced Accuracy : 0.5162
##
##   'Positive' Class : -1
##

```

Al final, el kernel polinomial ha sido el que mejor ha funcionado. El kernel lineal se comporta muy bien, pero el radial y el polinomial se comportan mejor.

Apartado d) (Bonus-3)

Ajustamos con validación cruzada sobre la variable *crim*.

```
cv.Boston <- cv.glmnet(as.matrix(Boston[,-1]), Boston[,1], nfolds = 5, alpha = 1)
```

El error de validación cruzada es

```
cv.Boston$cvm
```

```
## [1] 73.44629 69.69340 65.57450 62.15294 59.31050 56.94901 54.98697  
## [8] 53.35530 51.92575 50.57490 49.37295 48.37637 47.54303 46.84977  
## [15] 46.27829 45.80756 45.42499 45.11508 44.85812 44.63284 44.44202  
## [22] 44.28251 44.13463 43.99118 43.87275 43.77272 43.68583 43.61130  
## [29] 43.54863 43.49150 43.41694 43.35179 43.29459 43.22690 43.17175  
## [36] 43.12449 43.08471 43.03839 42.99640 42.94169 42.87580 42.82081  
## [43] 42.76855 42.72311 42.68992 42.66404 42.64521 42.63186 42.62700  
## [50] 42.62509 42.62654 42.62972 42.63239 42.63564 42.63938 42.64245  
## [57] 42.64645 42.65055 42.65177 42.65493 42.65777 42.66085 42.66415  
## [64] 42.66737 42.67058 42.67367 42.67688 42.68009 42.68274 42.68554  
## [71] 42.68788 42.69060 42.69247 42.69449 42.69618 42.69745
```

```
mean(cv.Boston$cvm)
```

```
## [1] 45.64565
```

Completar solución.

Ejercicio 3

Apartado a)

Ya tenemos cargado y separado el conjunto de datos en 80% training y 20% test.

Apartado b)

Vamos a ajustar un modelo de Bagging. Para ello usaremos el RandomForest y le diremos que use el total de características disponibles. El error de test lo mediremos siempre con el *MSE*.

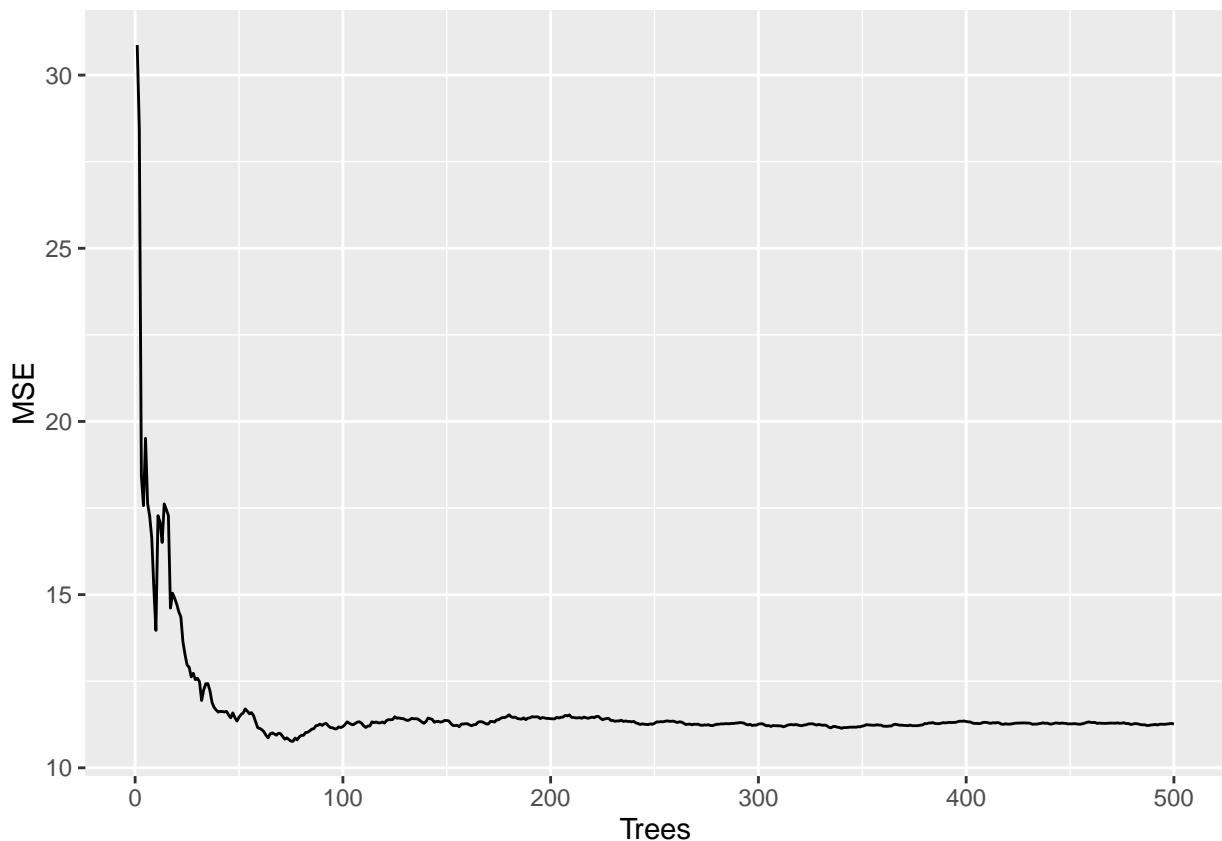
```
bagging <- randomForest(medv ~., data = Boston, subset = index, mtry = ncol(Boston)-1, importance = TRUE)  
bagging.prediction <- predict(bagging, newdata = Boston.test)  
bagging.error <- mse(medv[-index], bagging.prediction)
```

El error de test del modelo bagging es 5.9926212.

Apartado c)

Ahora vamos a ajustar un RandomForest.

```
randomFor <- randomForest(medv ~., data = Boston, subset = index, importance = TRUE)  
ggplot(data = data.frame(Trees = 1:randomFor$ntree, MSE = randomFor$mse), aes(x = Trees, y = MSE)) +  
  geom_line()
```



```
summary(randomFor)
```

```
##                                     Length Class  Mode
## call                           5    -none- call
## type                          1    -none- character
## predicted                     404   -none- numeric
## mse                           500   -none- numeric
## rsq                           500   -none- numeric
## oob.times                      404   -none- numeric
## importance                    26    -none- numeric
## importanceSD                  13    -none- numeric
## localImportance                0    -none- NULL
## proximity                     0    -none- NULL
## ntree                          1    -none- numeric
## mtry                          1    -none- numeric
## forest                         11   -none- list
## coefs                         0    -none- NULL
## y                             404   -none- numeric
## test                          0    -none- NULL
## inbag                         0    -none- NULL
## terms                        3    terms  call
randomFor.prediction <- predict(randomFor, newdata = Boston.test)
randomFor.error <- mse(medv[-index], randomFor.prediction)

randomFor.optimalntree <- randomForest(medv ~., data = Boston, subset = index, importance = TRUE,
                                         ntree = which(randomFor$mse == min(randomFor$mse)))
randomFor.optimalntree.prediction <- predict(randomFor.optimalntree, newdata = Boston.test)
```

```
randomFor.optimalntree.error <- mse(medv[-index], randomFor.optimalntree.prediction)
```

El número de árboles óptimo es 76. El error de test usando 500 es 5.9235047, y el error de test usando el óptimo es 6.2009486.

En este caso usar el óptimo no nos ha ofrecido mejora sobre el error de test.

Apartado d)

Ajustamos un modelo de regresión con Boosting. Como no tiene el mismo número de árboles por defecto que *randomForest*, lo ajustamos nosotros a 500, que es el parámetro por defecto del otro.

```
boosting <- gbm(medv ~ ., data = Boston.train, distribution = "gaussian", n.trees = 500)
boosting.prediction <- predict(boosting, Boston.test, n.trees = 500)
boosting.error <- mse(medv[-index], boosting.prediction)
```

El error de test es 40.6170962.

La diferencia con bagging y *randomForest* es notoria. Boosting se comporta mucho peor, dejando en el mejor lugar a RandomForest por muy poquito sobre Bagging. Esto es previsible, ya que *randomForest* tiene una diversificación que no tiene *Bagging*, este tiende a sobreajustar más.

Ejercicio 4

Apartado a)

Cogemos una muestra aleatoria de 800 elementos y lo usamos como training.

```
set.seed(123456789)
index <- sample(nrow(OJ), 800)
OJ.train <- OJ[index,]
OJ.test <- OJ[-index,]
```

Ajustamos un árbol con la variable *Purchase* como objetivo.

```
model.tree <- tree(Purchase ~ ., data = OJ.train)
```

Apartado b)

Veamos un resumen del árbol.

```
(model.tree.summary <- summary(model.tree))
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ.train)
## Variables actually used in tree construction:
## [1] "LoyalCH"      "PriceDiff"     "DiscMM"        "SalePriceMM"
## Number of terminal nodes:  10
## Residual mean deviance:  0.7312 = 577.7 / 790
## Misclassification error rate: 0.1713 = 137 / 800
```

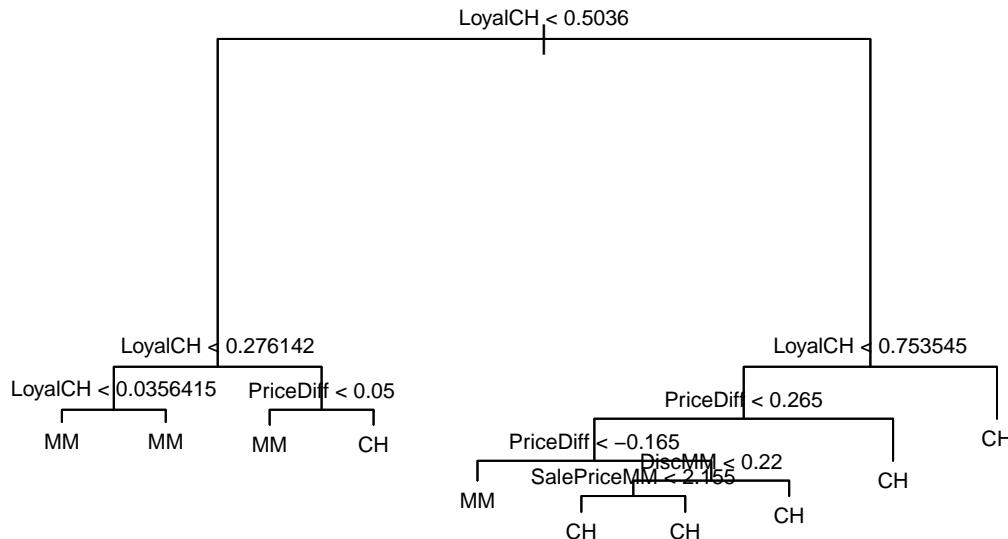
El número de nodos terminales es de 10, y tiene un error del 17.125%. Sin probar otros clasificadores, podemos afirmar que el árbol no es un buen clasificador para este problema, siguiendo la regla de que un clasificador

se considera bueno a partir del 90% de acierto. Las variables usadas han sido LoyalCH, PriceDiff, DiscMM, SalePriceMM.

Apartado c)

Dibujamos el árbol obtenido.

```
plot(model.tree, main="Classification tree")
text(model.tree, cex=.7)
```



La variable que más ganancia de información tiene es *LoyalCH*, los dos primeros niveles del árbol usan solamente esta variable. Además, hay algunas ramificaciones sin utilidad: Si $LoyalCH < 0.27$, no hace falta volver a distinguir para deducir *MM*, por ejemplo.

Apartado d)

Aplicamos el árbol a nuestros datos de test.

```

prediction.tree <- predict(model.tree, OJ.test, type = "class")
(error.test.tree <- confusionMatrix(prediction.tree, OJ.test$Purchase))

## Confusion Matrix and Statistics
##
##             Reference
## Prediction   CH   MM
##           CH 145   26
##           MM   15   84
##
##                   Accuracy : 0.8481
##                   95% CI : (0.7997, 0.8888)
##       No Information Rate : 0.5926
##       P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6805
## McNemar's Test P-Value : 0.1183
##
##       Sensitivity : 0.9062
```

```

##          Specificity : 0.7636
##      Pos Pred Value : 0.8480
##      Neg Pred Value : 0.8485
##          Prevalence : 0.5926
##      Detection Rate : 0.5370
## Detection Prevalence : 0.6333
##      Balanced Accuracy : 0.8349
##
##      'Positive' Class : CH
##

```

La matrix de confusión muestra todos los datos necesarios. El error de test es $1 - \text{Accuracy}$, 0.1518519

Apartado e)

Aplicamos la función cv.tree() a los datos de training y veamos qué hace.

```

model.cv.tree <- cv.tree(model.tree, K = 5)
model.cv.tree

## $size
## [1] 10 9 8 6 5 4 3 2 1
##
## $dev
## [1] 781.6995 762.5049 730.5119 734.8517 737.0078 784.3841 784.3841
## [8] 778.7347 1070.9272
##
## $k
## [1] -Inf 11.00549 13.51258 14.02312 17.78698 37.79043 38.60387
## [8] 46.64903 294.32466
##
## $method
## [1] "deviance"
##
## attr(),"class")
## [1] "prune"       "tree.sequence"

```

Para cada tamaño de árbol, calcula su error con validación cruzada. El mínimo error es el mínimo de dev . Lo veremos con una gráfica en el siguiente apartado.

Apartado f) (Bonus-4)

```

error.tree <- data.frame(size = model.cv.tree$size,
                           error = model.cv.tree$dev)

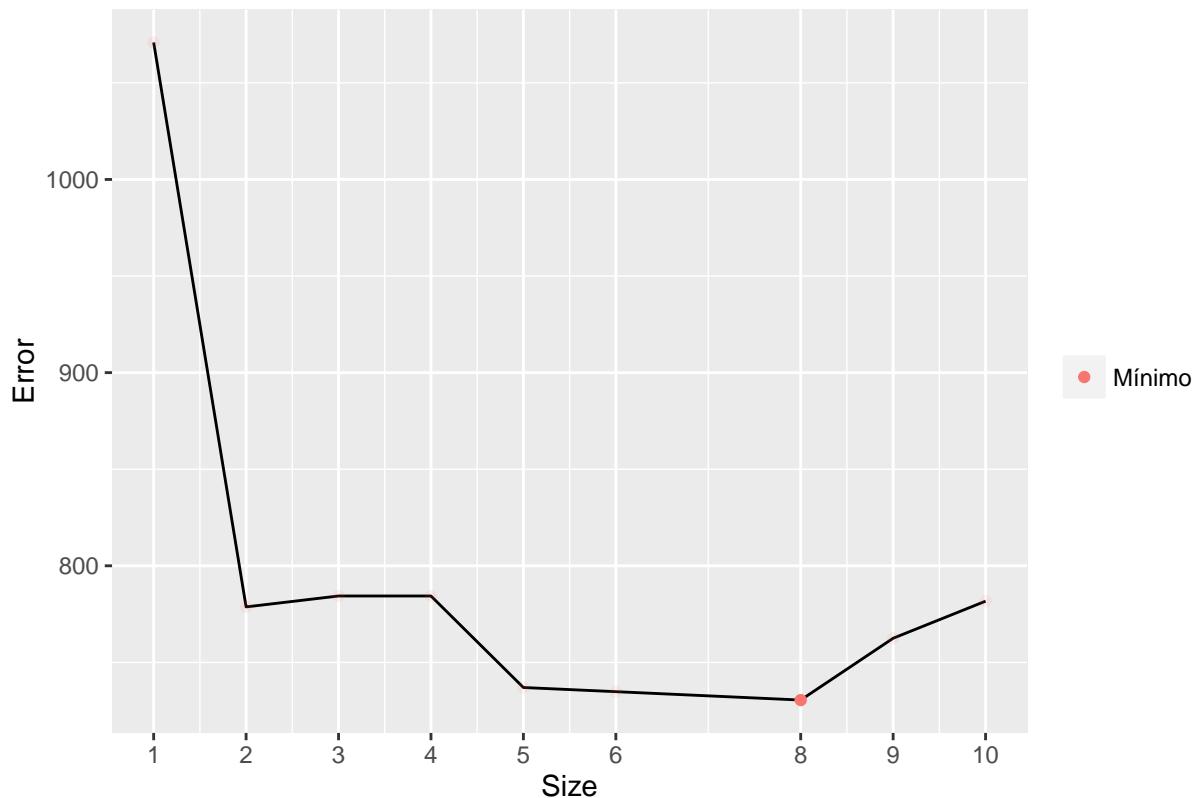
error.tree$alpha <- ifelse(error.tree$error == min(error.tree$error), 1, 0)

ggplot(error.tree, aes(x=size,y=error)) + geom_line() +
  geom_point(aes(colour = "Mínimo", alpha = alpha )) +
  scale_x_continuous(breaks = error.tree$size) +
  guides(alpha=FALSE) +
  theme(legend.title = element_blank()) +
  labs(title = "Cross Validation error",

```

```
x = "Size",  
y = "Error")
```

Cross Validation error



Podemos ver que el mínimo de error se alcanza en 8.